John von Neumann Institute for Computing

Comparing Two Parallel File Systems: PVFS and FSDDS

J. Buenabad-Chávez, S. Domínguez-Domínguez

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 507-514, 2006.

http://www.fz-juelich.de/nic-series/volume33

Comparing Two Parallel File Systems: PVFS and FSDDS

Jorge Buenabad-Chávez[a], Santiago Domínguez-Domínguez[a]

[a]Sección de Computación, CINVESTAV, Apartado Postal 14-740, México D.F., 07360, México.

1. Abstract

Parallel file systems are used to improve the performance of out-of-core parallel applications. The Parallel Virtual File System (PVFS) uses caching both to improve performance and to support logical views of data in order to simplify programming. The file system atop the data diffusion space (FSDDS) maps files onto an all-software distributed shared memory, and thus implicitly uses a relatively large cache. In this paper, we contrast the programming interface of PVFS and FSDDS and present some experimental results on their performance using two applications.

2. Introduction

Parallel file systems stripe file data across different I/O nodes so that data can be accessed in parallel on multiple, concurrent read/write operations to the same file. They are essential to improve the performance of out-of-core applications. First designs were targeted at massively parallel systems. Today the ubiquity of PC clusters and the availability of open/free designs and of other software tools have made their use and research wide-spread [2,3,5,7,9,11].

Parallel data access is a key factor for good performance, but is not the only one. Different applications manage different data structures in different ways [10,12]. In parallel applications, this implies a logical data partitioning among the processors which may, or may not, match the physical data partitioning (striping) of data across I/O nodes by a parallel file system [13]. This mismatch tends to increase the number of I/O operations, resulting in poor performance. Some I/O interfaces reduce the number of I/O operations through caching: reading/writing larger amounts of data than that requested by applications. Some interfaces also allow applications to specify a logical view of the data and access it accordingly. On a read, the interface reads all the data blocks that contain the logical data (possibly in parallel from different I/O nodes), shuffles the data according to the logical view and delivers it to the application. This also requires caching.

In this paper we compare two parallel file systems: PVFS and FSDDS. PVFS (Parallel Virtual File System) was designed for Linux clusters [2], and has gained general acceptance. It can be used with the MPI-IO interface which allows applications to access data according to logical views.

FSDDS stands for File System atop the Data Diffusion Space (DDS) [4]. DDS is another all-software distributed shared memory, and FSDDS supports file mapping onto its shared address space. DDS manages some memory as a cache in each node to dynamically map shared data; FSDDS thus benefits of a relatively large cache.

In this paper we contrast the programming interface and present some experimental results on the performance of PVFS and FSDDS. In Sections 3 and 4 we present background to PVFS and FSDDS, respectively. In Section 5 we compare their performance running 2 parallel applications on different processor counts. We conclude in Section 6.

3. The Parallel Virtual File System

The Parallel Virtual File System (PVFS) was designed for Linux clusters. It supports several APIs to access PVFS files, can be mounted as a UNIX file system (*ls*, *cp* and *rm* commands work on PVFS files), and is rather easy to install and use [2]. It is open/free software.

File data in PVFS is striped across different I/O nodes based on three metadata parameters: *pcount* specifies the number of I/O nodes across which data is striped; *base* specifies the I/O node where the striping begins; and *ssize* specifies the stripe size. PVFS handles default values for the striping metadata, which the user can change for each file. File data and metadata are stored in files in the local file system in each node, both for simplicity and for portability.

PVFS is organised as a client/server system. Server nodes have hard disk space and are those across which file data is striped; they are called I/O nodes. Client nodes are those where application processes run, issuing read/write requests to I/O nodes; they are called compute nodes. PVFS software allows each node to be either a compute node or an I/O node, or both. Application processes are linked to a PVFS library which allows them to communicate with I/O nodes through TCP.

3.1. PVFS APIs

PVFS files can be accessed with different APIs: a native PVFS API, the UNIX/POSIX API [6], and the MPI-IO interface [8]. The native API offers similar functions to the UNIX one for accessing files. It also supports functions to access noncontiguous data in a file in a single call. The MPI-IO interface offers typical functions for accessing a file, but also offers functions to define logical views, and collective I/O functions which may, or may not, be based on a logical view.

Figure 1 shows in C language code the main points involved in the definition and use of a logical data view using MPI-IO. The logical view is that, for an $N \times N$ matrix and p processors, processor 0 uses only the first N/p elements in each row, processor 1 uses only the second N/p elements in each row, and so on. (This view is useful to implement a matrix multiplication algorithm, $C = A \times B$, where each processor computes a partial value of each element in matrix C; before reading A, each processor will have read N/p rows of matrix B; see Section 5.)

```
MPI_Datatype newtype;                                  /* the new logical view */
int ndims=2,

array_of_gsizes[0]   = N;                              /* size of each dimension*/
array_of_gsizes[1]   = N;
array_of_distribs[0] = MPI_DISTRIBUTE_BLOCK;           /* divide rows by block */
array_of_distribs[1] = MPI_DISTRIBUTE_NONE;            /* do not divide columns */
array_of_dargs[0]    = MPI_DISTRIBUTE_DFLT_DARG;       /* block = rowsize/processors */
array_of_dargs[1]    = MPI_DISTRIBUTE_DFLT_DARG;       /* not applicable */
array_of_psizes[0]   = 0;                              /* compute rowsize/processors */
array_of_psizes[1]   = 1;                              /* do not compute */
MPI_Dims_create( nprocs, ndims, array_of_psizes);      /* divide rows/nprocs */
MPI_Type_create_darray(nprocs, myrank, ndims,          /* define view */
                       array_of_gsizes, array_of_distribs, array_of_dargs,
                       array_of_psizes, MPI_ORDER_C, MATRIX_MPI_TYPE, &newtype);
MPI_Type_commit( &newtype );                           /* logical view handle */
MPI_Type_size( newtype, &bufcount );

MPI_File_open( MPI_COMM_WORLD, ..., &f );              /* use view on file */
MPI_File_set_view( f, 0, MATRIX_MPI_TYPE, newtype, "native", MPI_Info );
MPI_File_read_all( f, readbuf, N, MATRIX_MPI_TYPE, &status );
MPI_File_close( &f );
```

Figure 1. Logical view for a PVFS file using MPI-IO.

A logical view is defined using three (type int) arrays: *array_of_gsizes* holds the size of each dimension in the array upon which the logical view is defined; *array_of_distribs* says whether a dimension is distributed and how (NONE, BLOCK, or CYCLIC); *array_of_dargs* specifies the size of the distribution unit (BLOCK may use the default SizeOfDimension/NumberOfProcessors; CYCLIC requires a unit size); and *array_of_psizes* specifies programmer-defined number of elements to distribute to each processor if defaults are to be ignored. The view is then created and a *newtype* is defined with it. The view is then attached to a file and used.

4. FSDDS: A File System atop the Data Diffusion Space

FSDDS is a parallel file system for the Data Diffusion Space (DDS), an all-software distributed shared memory for PC Clusters. FSDDS supports typical functions to access files but also allows mapping files onto the shared address space of DDS.

4.1. DDS

DDS supports a shared address space for parallel applications running on distributed memory platforms under the SPMD (Single Program Multiple Data) model [1]. The size of the shared address space can be up to 2^{64} bytes, either on 32-bit or on 64-bit architectures. Shared data diffuses in the memory of each processor using the data, or in the disk space of each processor if need be, under a multiple-readers-single-writer protocol.

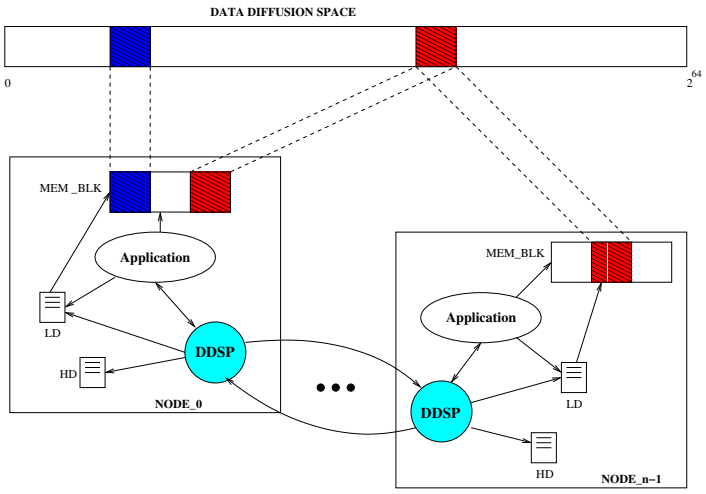


Figure 2. The DDS Architecture.

Figure 2 shows the DDS architecture. In each node runs an application process and a DDSP process. The data diffusion space is *extra* to the address space of each process running a parallel application. Shared data is dynamically mapped into the address space of whichever application process is using the data. A *local directory* is used to determine if a shared data item is already mapped (looking up its address). If it is not mapped, the sibling DDSP process sends a request to its *home directory* (HD) node, which is identified using the hash function *item-address modulo number-of-nodes*. A home directory holds the location (node) and state information (exclusive, shared, master shared) of a data item. DDSP processes communicate through TCP sockets.

4.2. File Management based on DDS

FSDDS provides applications with file management based on DDS [4]. As in other parallel file systems, in FSDDS: file data is striped across different I/O nodes, metadata is used to describe the striping, and compute nodes and I/O nodes interact as clients and servers, respectively. Data striping and metadata management are based on those of PVFS. Also as in PVFS, a node can be either a compute node or an I/O node.

When a file is opened under FSDDS, it is automatically mapped onto the shared address space of DDS. The first tera byte of DDS is reserved for shared data in arrays (i.e., not file data); the following tera bytes are used one for each file that is open. Thus files and shared data are accessed in the same way, and under the multiple-readers-single-writer protocol exerted by DDS. (The API of FSDDS is described in Section 4.3.)

For each data item in a file, its I/O node is also its home directory node; and these two roles are carried by the DDSP process in that node. However, recall that the home node of a shared (array) data item is determined with the hash function *item-address modulo number-of-nodes* , while the I/O node of a file data item is determined through metadata. Whether the hash function or metadata is used depends on the address of each data item: the address of file data items will be equal to or greater than 1 tera byte because of the mapping of files described above. This distinction also entailed some changes to the DDS replacement policy, as described below, in order to improve performance.

In each node, when the memory becomes full, a less recently used item is chosen and an action is taken depending on its status. If the status is *shared*, the item is just discarded. If it is *exclusive* and the item belongs to an array (not a file), the item is swapped onto a local temporary file. If it is *exclusive* and the item belongs to an FSDDS file, the item is sent to its I/O node, unless the *current* node is that I/O node, in which case the item is just swapped out onto its FSDDS local file. If the status of the item is *master shared* and the item belongs to an array, the item is sent to its home node, unless the current node is the home node, in which case the item is sent to another node chosen randomly. If the status of the item is *master shared* and the item belongs to an FSDDS file, the item is swapped out onto a local temporary file, or its FSDDS local file if the current node is its I/O node.

4.3. API

Figure 3 shows the addition of two matrices, $C = A + B$, using an FSDDS file for each matrix. DDS_Init is the first DDS function that must be called; it establishes communication with the sibling DDSP process, which allocates the (DDS) cache memory to store shared data, and initialises the local directory and the home directory. A file is opened/created with DDS_Open; then its data is automatically mapped onto DDS. Each processor computes $ROWS/nprocs$ rows. Before accessing data, each processor must gain access to it, through calling DDS_Write or DDS_Read. When these procedures return, the relevant data is already in the processor memory, and will remain there until the corresponding $DDS\_UnWrite$ or DDS_UnRead is issued.

Data is actually accessed through pointers held in the array dds_shmem, and the variables off_fa, off_fb and off_fc, which are associated to the file descriptors, and are *locally* shared between the DDSP process and the application process. Those variables are updated by DDSP according both to the address of the data requested with DDS_Read or DDS_Write, and to the shared address allocated to the array when it was opened.

5. Performance Evaluation

To evaluate the performance of PVFS and FSDDS we ran two applications on a 16-node PC cluster using different numbers of processors. Each node in the cluster consisted of one Intel Celeron 1.7

```
int  fa, fb, fc;                          /* Definition of file descriptors */
main(int argc, char **argv) {
  DDS_Init(NULL, NULL, mynod);            /* initializing DDS*/
  fa = DDS_Open("matrixA", O_RD | O_CREAT, NULL);
  fb = DDS_Open("matrixB", O_RD | O_CREAT, NULL);
  fc = DDS_Open("matrixC", O_RDWR | O_CREAT, NULL);

  rows = ROWS/nprocs;                     /* Each processor computes ROWS/nprocs rows. */
  offset = myid * (ROWS/nprocs);
  for (r=0; r < rows; r++){
    i = r + offset;
    DDS_Read( fa, i*COLUMNS, COLUMNS);    /* gaining access */
    DDS_Read( fb, i*COLUMNS, COLUMNS);    /* to shared data */
    DDS_Write(fc, i*COLUMNS, COLUMNS);
    for (j=0; j<NCA; j++){                /* using shared data */
      (dds_shmem[off_fc+i])[j] = (dds_shmem[off_fa+i])[j] +
                                 (dds_shmem[off_fb+i])[j] ;
    }
    DDS_UnWrite(fc, i*COLUMNS, COLUMNS);
    DDS_UnRead(fa, i*COLUMNS, COLUMNS);
    DDS_UnRead(fb, i*COLUMNS, COLUMNS);
  }                                       /* Close files */
  DDS_Close(fa);  DDS_Close(fb);  DDS_Close(fc);
  DDS_Finalize();                         /* Finalize DDS*/
}
```

Figure 3. FSDDS programming model example: matrix multiplication.

GHz processor, 512 MB RAM memory, and a hard disk. Hard disks are of different make and storage capacity (4 and 8 GB). All nodes were interconnected through a 3COM Fast Ethernet switch with 48 ports. The operating system was Linux RedHat 9.0.

The runs with PVFS used the MPI-IO interface to issue collective I/O operations. Both the runs with PVFS and the runs with FSDDS used data stored in files. In all runs, all the nodes function both as compute nodes and as I/O nodes (data is physically partitioned among all nodes). The striping of files was different for each application and is described below.

5.1. Application 1: Matrix Multiplication

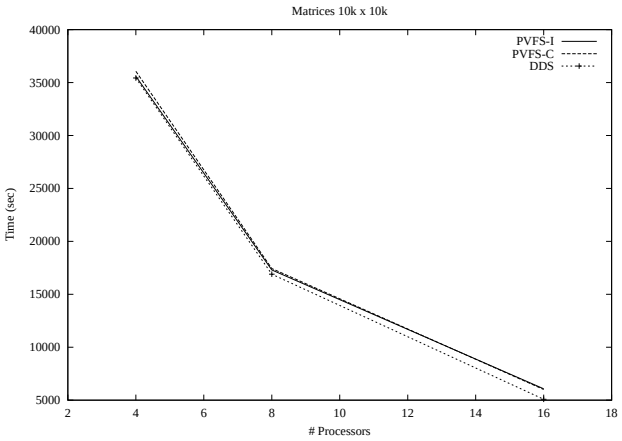
Our first application is a matrix multiplication algorithm (MM), $C = A \times B$. For $N \times N$ matrices and p processors, processor 0 computes the partial result of each element in C using the first N/p elements of the corresponding row in A and the first N/p elements of the corresponding column in B; processor 1 does the same using the second N/p elements ..., etc. Each processor computes the partial results of an entire row in C in one go, and then adds them to the actual elements in C. All processors start computing the first row in C. This algorithm matches the data access pattern with the physical data partitioning in secondary storage (by rows, according to the row-major order of the C language), and allowed us to validate the synchronisation needed to maintain data coherence on writing matrix C.

In all our experiments, each matrix is 10000×10000 long type elements (4 bytes each), is stored in a file by rows, and each file is striped across p processors (nodes). Under PVFS with *individual* (non-collective) read/write operations (PVFS-I), and under FSDDS, the stripe size in all matrices was N/p rows. We also ran a PVFS version where each processor uses collective read/write operations (PVFS-C), using a stripe of size N/p data elements for matrix A only.

Figure 4.a shows the number of read and write requests under PVFS-I, PVFS-C and FSDDS. Both PVFS versions incur the same number of read requests because they differ only in the kind of read they use: each read operation is for the same amount of data. Under FSDDS, some reads for rows in

PVFS-I/C			FSDDS	
Procs	Reads	Writes	Reads	Writes
4	12500	2500	5000	2500
8	11250	1250	2500	1250
16	10625	625	1250	625

(a) Read and write requests.



(b) Execution time.

Figure 4. MM under PVFS and FSDDS.

matrix A were satisfied from copies in other memory nodes. All versions incur the same number of write requests because they use the same write unit, a row. Under PVFS-I/C, each row is written out by one processor once it is computed. Under FSDDS, rows in matrix C are held in memory as much as possible; hence some of them are written out only when the file is closed.

Figure 4.b shows the execution time of MM under PVFS-I, PVFS-C and FSDDS. All versions show about the same execution time, even though under FSDDS less than half read requests are incurred. The reason for this is data caching. All processors access each row in matrix A starting at the first row, and once they finish computing the partial results of the corresponding row in C, they discard it; in contrast, each processor holds the rows of matrix B it uses throughout the computation. That is, each processor is accessing each row in each matrix into its memory only once, both under PVFS and under FSDDS. This implies that in PVFS some reads where satisfied from copies in main memory too, most likely from the cache of I/O nodes. FSDDS shows slightly better performance, on 8 and 16 processors, because PVFS versions used $MPI\_Gather()$ to collect all the partial results for a row in matrix C, and thus incur synchronisation cost, more so the larger the number of processors. Under FSDDS, each processor writes exclusively its partial results as they gain access to each row.

5.2. Fast Fourier Transform

Our second application applies the Fast Fourier Transform (FFT) to restore degraded or defocused images. For an image of $N \times N$ pixels, a matrix of size $N \times N \times 8$ (float type) bytes is used. From this matrix, an *images matrix* is created which corresponds to an autocorrelation process. The images matrix contains $M \times M$ images, where $M = 2N$, and is of size $2N \times 2N \times (N \times N) \times 8 = (N^4) \times 32$ bytes. To the images matrix, our application applies the FFT as follows. Processor 0 applies the FFT to the first M/p rows and to the first M/p columns (of images) along rows in each image matrix (1st), along columns in each image matrix (2nd), jumping through rows in different image matrices (3rd), and jumping through columns likewise (4th); processor 1 applies the FFT to the second M/p rows and to the second M/p columns (of images), ..., and so on. Figure 5 shows an images matrix for $N = 2$, and its partitioning for 4 processors.

We ran FFT on 4, 8 and 16 processors, both under PVFS (non-collective) and under FSDDS. In all

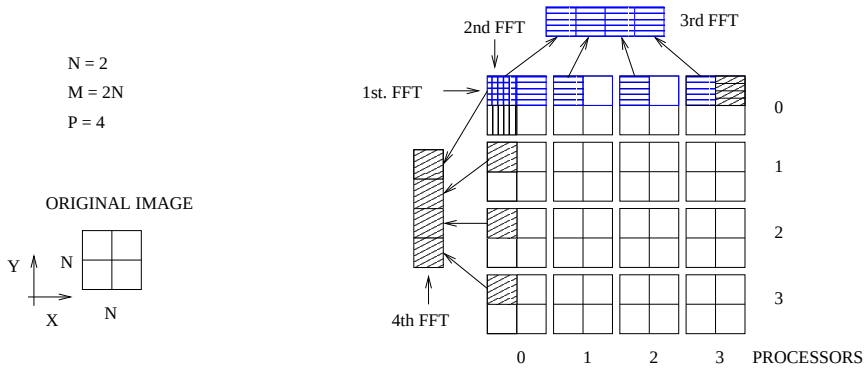


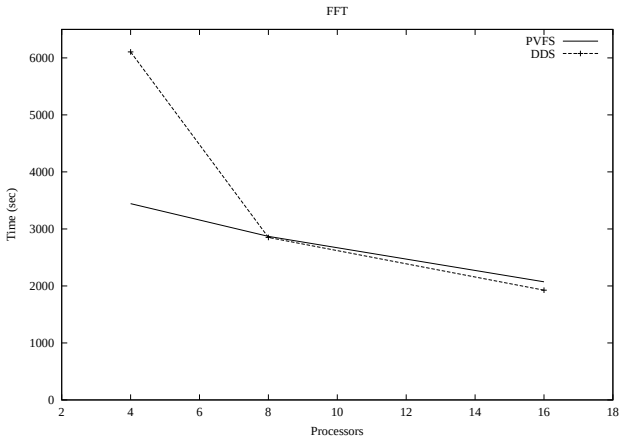
Figure 5. FFT: The images matrix for $N = 2$, and its partitioning for 4 processors.

runs, the stripe size was M/p rows of images, and the images matrix was of size $((128)^4) \times 32 = 8$ GB, and could not be held entirely in memory in all processor-count configurations.

Figure 6.a shows the number of I/O requests per processor. In each processor-count configuration, the number of I/O requests under FSDDS was smaller than under PVFS because, under FSDDS, some reads were satisfied from copies in other memory nodes, and because writes occurred in memory copies which were committed to disk only when there was a shortage of memory or until the file was closed.

	PVFS		FSDDS	
Processors	Reads	Writes	Reads	Writes
4	49152	49152	48729	48732
8	24576	24576	23638	23638
16	12288	12288	9934	9934

(a) Read and write requests.



(b) Execution time.

Figure 6. FFT under PVFS and FSDDS.

Figure 6.b shows the execution time of FFT, both under PVFS (non-collective) and FSDDS, on 4, 8 and 16 processors. On 4 processors, even though the number of reads and writes under FSDDS was smaller than the number of reads and writes under PVFS, the execution time under FSDDS was much greater because the available memory was relatively small, and thus the replacement policy of FSDDS was exerted frequently. On 8 and 16 processors, the larger memory available meant less use of the replacement policy, improving performance.

6. Conclusions and Future Work

We have outlined the use of PVFS and FSDDS, and presented some experimental results on their performance. Files in PVFS can be accessed rather simply through a UNIX-like interface; the MPI-IO interface can also be used to define logical data views. The use of views is not simple, however, even though there is some logic behind it. FSDDS interface is quite cumbersome, because DDS internal data structures are exposed. We are working on the design of an extension to the C language and its preprocessor to avoid the use of the DDS interface entirely. We envisage parallel applications will use arrays and files as a shared memory; the preprocessor will issue the corresponding DDS_Read-DDS_UnRead and DDS_Write-$DDS\_UnWrite$ pairs, and the *reference* to each shared data item based on DDS internal data structures.

The performance of PVFS and FSDDS was similar except for our second application (FFT) on 4 processors. On this application/configuration, the replacement policy of DDS had an adverse effect because the amount memory was relatively small. However, the data caching of DDS does work in general, as can be seen from the slightly better performance that FSDDS shows on 8 and 16 processors (3-10%). We are working on using different replacement policies and data striping methods to improve the performance of FSDDS.

References

[1] Jorge Buenabad-Chávez and Santiago Domínguez-Domínguez: The Data Diffusion Space for Parallel Computing in Clusters. In Proceedings of Euro-par 2005 (LNCS 3648) 61–71.

[2] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur: PVFS: A parallel file system for linux clusters. In Proceedings of the 4th Annual Linux Showcase and Conference 317–327. 2000.

[3] Peter F. Corbett and Dror G. Feitelson: The Vesta parallel file system. ACM Transactions on Computer Systems, 14 (3) 225–264. 1996.

[4] Santiago Domínguez-Domínguez, Jorge Buenabad-Chávez: Distributed Parallel File System for I/O Intensive Parallel Computing on Clusters. In Proceedings of International Conference on Electrical and Electronics Engineering and X Conference on Electrical Engineering, ICEEE/CIE2004, Acapulco, Guerrero, México, September 8-10. 2004.

[5] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal: PPFS: A high performance portable parallel file system. In Proceedings of the 9th ACM International Conference on Supercomputing 385–394. 1995.

[6] IEEE/ANSI Std. 1003.1: Portable operating system interface (POSIX)-part1: System application program interface (API) [C Language]. 1996.

[7] Florin Isaila and Walter F. Tichy: Clusterfile: a flexible physical layout parallel file system. Concurrency and Computation, 15 (7/8) 653–679. 2003.

[8] MPI-IO: A Parallel File I/O Interface for MPI, http://www.mpi-forum.org/docs/docs.htm. 1997.

[9] Nils Nieuwejaar and David Kotz: The Galley parallel file system. In Proceedings of the 10th ACM International Conference on Supercomputing 374–381. 1996.

[10] Ron Oldfield and David Kotz: Applications of Parallel I/O. Technical Report PCS-TR98-337. Department of Computer Science, Dartmouth College, Hanover.

[11] Ron Oldfield and David Kotz: Armada: A parallel file system for computational grids. In Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid 194–201. 2001.

[12] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best: Characterizing parallel file-access patterns on a large-scale multiprocessor. In Proceedings of the Ninth International Parallel Processing Symposium 165–172. 1995.

[13] Huseyin Simitci and Daniel Reed: A comparison of logical and physical parallel I/O patterns. The International Journal of High Performance Computing Applications, 12 (3) 364–380. 1998.